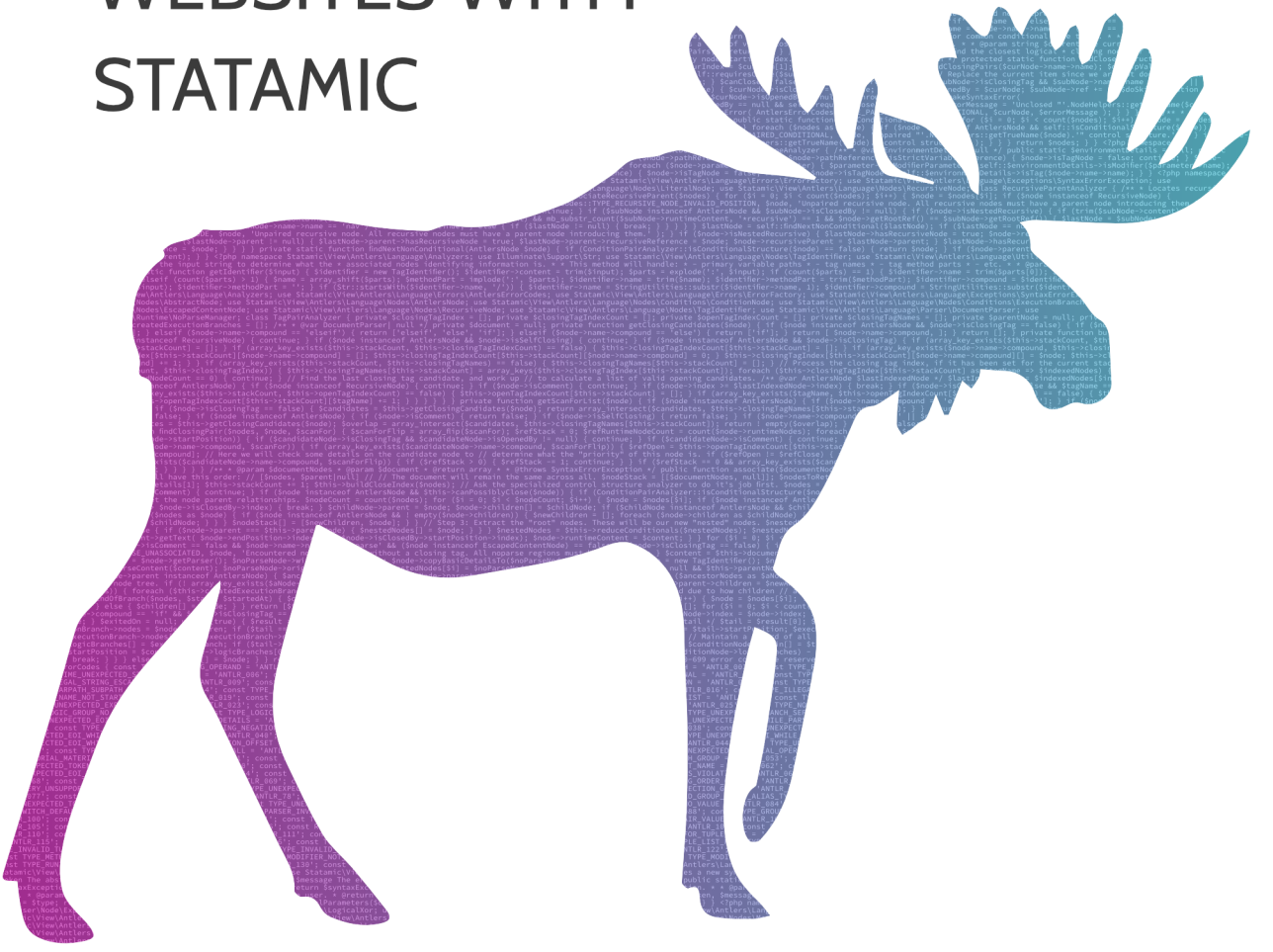
# ANTLERS

## BUILDING BEAUTIFUL WEBSITES WITH STATAMIC



J. KOSTER

# Antlers

## Building Beautiful Websites with Statamic

Johnathon Koster

This book is for sale at http://leanpub.com/antlers

This version was published on 2023-05-23


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

*L.S.*

# Contents

# Statamic, Laravel, and PHP Version

Throughout this book, we will explore many different concepts and ideas through many code examples. PHP 8 was used throughout the writing process, but the code examples can be adapted to any modern PHP version.

The contents of this book assume at least Statamic 3.4.6 and at least Laravel 8.48.0.

All Antlers code examples were written using the Runtime Antlers parser.

# Symbols Used in This Book

We will encounter a small number of icons throughout this book. These icons call out additional information and potential pitfalls or direct you to other time-saving resources.

Look for this icon in the text to learn about any appendices that provide additional reference material or context for a given section.

This icon alerts you to additional information that you may find interesting or provides further context around ideas to consider when implementing them in your projects.

This icon calls out information that may be technically correct for a given situation. Still, there may be issues that lead to common bugs.

This icon accompanies tips or additional information to help explain concepts in the surrounding text.

This icon accompanies installation instructions for third-party addons or packages discussed in the surrounding text.

# 1. An Antlers Primer

Welcome to Antlers, the templating language designed to make your Statamic development journey more accessible and efficient. In this book, we will take you through the ins and outs of Antlers, from the basics to the advanced features, so you can easily create unique and beautiful websites. As we progress through the book, we will focus on what we can accomplish with the Antlers language and selecting Antlers Tags and modifiers along the way. Due to the highly configurable nature of Statamic itself, we will refrain from overly specific site structures, fieldset configurations, and similar as we explore what is possible.

Antlers has a rich history, with its roots dating back over ten years to the original language, Lex, which was created for PyroCMS. Over the years, Antlers has evolved significantly, with four major releases in Statamic alone, as well as drawing inspiration from newer languages such as Laravel's Blade.

As you dive into this book, you will discover how Antlers has been precisely crafted to simplify the process of building Statamic websites. Whether you are a beginner or an experienced developer, this book will provide additional knowledge to help make the most of Antlers and create stunning websites that are fast, reliable, and easy to maintain.

Antlers itself can be broadly broken into three main ideas:

- Tags: Tags are a powerful concept allowing Antlers to interact with backend PHP code and generally introduce *new* data into a template.
- Variables: Variables allow us to reference existing data within a template to render it on screen, change it in exciting ways, or supply it to other language features, such as Tags or modifiers.
- Modifiers: Modifiers generally act upon existing data and allow us to transform, perform operations on, or move complex logic to our backend PHP code.

Our Antlers template will appear within other structured documents, such as HTML or XML, and our template code will appear between pairs of curly braces. As an

example, let's take a look at the following template that renders a simple HTML document with a dynamic title:

**Figure 1.1**

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>{{ title }}</title>
5    </head>
6    <body>
7
8    </body>
9  </html>
```

The active part of our template in **Figure 1.1** can be found on line 4 and is an example of displaying the value of a variable using Antlers.

In the following example, we can see an example of an Antlers tag pair, which will return all entries from a collection of pages and display their title:

**Figure 1.2**

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>{{ title }}</title>
5    </head>
6    <body>
7      {{ collection:pages }}
8        {{ title }}
9      {{ /collection:pages}}
10   </body>
11 </html>
```

We can find our tag pair between lines 7 and 9 in **Figure 1.2**. The collection tag will return entries, or individual pieces of content, from the collections we specify. In our

example, we are asking it for entries within a "pages" collection. Inside the tag pair, on line 8, we can see we are using the `{{ title }}` variable as we did on line 4.

If the pages returned from the collection tag have a title, that will be used on line 8. Suppose a page does not have a title; the same value will that was rendered on line 4 will be used instead. This behavior is a simple example of a powerful concept known as the Cascade; it allows access to the closest data first, within the current nesting level, and continuously works upwards until it finds a match.

Tag pairs are easily the most powerful feature within the Antlers templating language and serve many purposes. As we saw in **Figure 1.2**, we can use them to interact with the backend CMS or custom PHP code, but their primary function is to loop over arrays or provide access to key/value pairs in the case of associative arrays.

These many roles may be confusing at first, but it becomes more apparent when we realize that *most* tags return an array of items or a key/value pair of additional data we can use when rendering content to the screen.

As a quick example, let us imagine we already have an array of entries stored inside a `pages` variable. We can loop over these pages using a tag pair and access all of the data stored inside:

**Figure 1.3**

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>{{ title }}</title>
5    </head>
6    <body>
7      {{ pages }}
8        {{ title }}
9      {{ /pages}}
10   </body>
11 </html>
```

The syntax differences are minimal when we compare the template code in **Figure 1.2** to **Figure 1.3**; the primary difference is *where* we got the data. In the case of

**Figure 1.2**, we got the data from the results of the collection Tag, and in **Figure 1.3**, we assume it has come from somewhere else.

We can also use modifiers to perform some action on *existing* data, and they appear after a variable, separated by a vertical pipe (|) character. For example, we can use the `upper` modifier to transform text values into their uppercase version:

**Figure 1.4**

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    {{ collection:pages }}
      {{ title | upper }}
    {{ /collection:pages}}
  </body>
</html>
```

# 1.1 Tags vs. Variables

So far, we have seen examples of a tag pair that invokes some backend CMS logic with the collection Tag and a tag pair that loops over some existing data. When comparing the two examples in the previous section, it may be challenging to determine what a Tag and variable are just by looking at a template. Recognizing what is likely to be a Tag or variable comes with time and experience building Statamic sites.

## Core Tag Names

Appendix A: Core Statamic Tag Names contains a list of all Tags provided by Statamic. This reference may be helpful to quickly see what is available, as well as check what may already be in use.

When rendering templates, Antlers will prioritize variables over Tags when a name collision exists; this can lead to probably the most infamous Statamic error message

`modifier [from] not found`. To help explain this behavior, we can force this to happen with the following template:

**Figure 1.5**

```
{{ collection = ['one', 'two', 'three']; }}

{{ collection from="pages" }}

{{ /collection }}
```

One line 1, we create a new variable named `collection` to force a variable/Tag name collision. When we attempt to render our template, we will receive the following error message:

**Figure 1.6**

```
Statamic\Modifiers\ModifierNotFoundException

Modifier [from] not found
```

While we have forced this to happen in our example template, this can also occur naturally on mounted pages or if Tags inject variable names that match other Tags. Luckily, we can quickly resolve this by forcing Antlers to always consider something a Tag by prefixing it with `%`:

**Figure 1.7**

```
{{ collection = ['one', 'two', 'three']; }}

{{ %collection from="pages" }}

{{ /%collection }}
```

An important thing to note is that when forcing something to be a Tag pair, we must also include the `%` prefix on the closing tag. We can also force tag behavior on non-tag pairs using the same prefix:

**Figure 1.8**

```
{{ %partial src="template/path" /}}
```

# 1.2 Modifiers

Modifiers allow us to perform operations or change data we already have. When working with modifiers, there are two syntax styles: array and shorthand.

The array-style syntax is similar in appearance to Tag parameters and is typically seen used together with variable tag pairs. As an example, we can use array-style syntax to reverse the order of an array like so:

**Figure 1.9**

```
{{ _data = ['One', 'Two', 'Three']; }}

{{ _data reverse="true" }}
  {{ value }}
{{ /_data }}
```

Our template in **Figure 1.9** would produce the following output:

**Figure 1.10**

```
Three
Two
One
```

Historically, shorthand syntax has been used on non-tag pairs, such as with text or string data:

**Figure 1.11**

```
{{ title | upper }}
```

However, we can also use the shorthand syntax on tag pairs. For example, we can rewrite our array reversing example using shorthand syntax like so:

**Figure 1.12**

```
{{ _data = ['One', 'Two', 'Three']; }}

{{ _data | reverse }}
  {{ value }}
{{ /_data }}
```

Attempts will be made throughout this book to always use the shorthand syntax, even on tag pairs, as this helps to differentiate between tags and variables visually.

Antlers will utilize the value that appears to the left of the modifier in the template when evaluating modifiers. In the case of chained modifiers, Antlers will supply the results of one modifier to the following modifier. For example, the first `reverse` modifier will receive the values from our `_data` array. However, the second `reverse` modifier will receive reversed output from the first modifier and reverse that.

**Figure 1.13**

```
{{ _data = ['One', 'Two', 'Three']; }}

{{ _data | reverse | reverse }}
  {{ value }}
{{ /_data }}
```

After both modifiers have been evaluated, our template would output the following:

**Figure 1.14**

```
One
Two
Three
```

## Modifiers and `null` Values

Antlers will stop evaluating modifiers whenever it encounters a `null` or non-existent value. If multiple modifiers are chained, and a modifier returns `null`, the remaining modifiers will be skipped, and the entire chain will return `null`.

Most modifiers also accept parameters, such as the `explode` modifier, which can be used to split a text value on some character sequence. We can supply parameters to modifiers much like we do when calling functions in PHP:

**Figure 1.15**

```
{{ _stringValue = 'one,two,three' /}}

{{ _stringValue | explode(',') }}
  {{ value }}
{{ /_stringValue }}
```

Our template in **Figure 1.15** would produce the following output:

**Figure 1.16**

```
one
two
three
```

To pass multiple parameters to a modifier, we separate them with a comma:

**Figure 1.17**

```
{{ _stringValue = 'The quick brown fox' /}}

{{ _stringValue | replace('quick', 'extremely fast') }}
```

Which produces:

**Figure 1.18**

```
The extremely fast brown fox
```

# 1.3 Comments

Like with most templating or programming languages, Antlers supports comments. Comments are created by wrapping a block of template code in a `{{#` and `#}}` pair.

**Figure 1.19**

```
{{#

  This will not be evaluated:

  {{ collection:pages }}
    {{ title }}
  {{ /collection:pages }}

#}}
```

Antlers does *not* support inline comments when already inside a pair of curly braces.

# 1.4 Escaping Antlers Code

Escaping code is an essential technique in template languages like Antlers. It allows you to include raw code or characters in your templates without them being parsed

and executed by the template engine. In Antlers, escaping code is simple and can be achieved using the `@` symbol notation and the `noparse` Tag.

The `noparse` Tag is convenient to prevent Antlers from processing large parts of your template. You can wrap the code you want to escape with the `noparse` Tag, instructing Antlers to treat the content between the tags as plain text. This is particularly useful when you include example code or characters that might interfere with the parsing process or when writing template code for front-end libraries with identical syntaxes.

An example of the `noparse` Tag in an Antlers template may look similar to **Figure 1.20:**

**Figure 1.20**

```
{{ noparse }}
<ul>
  <li v-for="item in items">
    {{ item.message }}
  </li>
</ul>
{{ /noparse }}
```

In this example, the content between the Tag pair will be output as literal text.

Another common method to escape code in an Antlers template is to prefix curly braces with the `@` character; this will instruct Antlers to emit a literal curly brace instead. When we are working outside of curly braces, we only need to escape the first opening curly brace like so:

**Figure 1.21**

```
<ul>
  <li v-for="item in items">
    @{{ item.message }}
  </li>
</ul>
```

Our template in **Figure 1.21** would produce the following output:

**Figure 1.22**

```
<ul>
  <li v-for="item in items">
    {{ item.message }}
  </li>
</ul>
```

However, when working inside existing curly brace pairs, we will need to escape each instance of the opening and closing braces we don't want to be parsed by Antlers; this is common when constructing dynamic strings or supplying literal braces inside tag parameters:

**Figure 1.23**

```
{{# Outputs {STRING CONTENT} #}}
{{ '@{String Content@}' | upper }}

{{ tag:method parameter="Value with braces @{@}" }}

{{ /tag:method }}
```

One last common technique is to create a new partial that does not contain the `.antlers.html` file extension, and include that in the template using the `partial` Tag. Files not beginning with an Antlers file extension will not be parsed as Antlers, and their contents will be returned.

In `resources/views/_partial_name.html`:

**Figure 1.24**

```
<ul>
  <li v-for="item in items">
    {{ item.message }}
  </li>
</ul>
```

In `resources/views/template.antlers.html`:

**Figure 1.25**

```
{{# Include our HTML file containing curly braces. #}}
{{ partial:partial_name }}
```

# 1.5 Self-Closing Tags

The tag pairing algorithm will attempt to pair the first opening tag candidate it can with the *furthest* matching closing tag it can. This behavior is to accommodate situations like the following, where there may be internal ambiguity:

**Figure 1.26**

```
{{ collection from="pages" }}
  {{ collection:handle }}
{{ /collection }}
```

The internal pairing behavior can largely be ignored during normal day-to-day development, but it may cause unexpected output in situations like the following:

**Figure 1.27**

```
1 {{ partial src="one" }}
2 {{ partial src="two" }}
3   <!-- Slot content -->
4 {{ /partial }}
```

In **Figure 1.27**, Antlers will create a Tag pair from the opening `partial` tag on line 1 and the closing tag on line 4. We can influence this behavior by using self-closing Antlers tags.

Self-closing Antlers tags end with `/}}`, and the tag pairing algorithm will skip over these entirely:

**Figure 1.28**

```
1 {{ partial src="one" /}}
2 {{ partial src="two" }}
3   <!-- Slot content -->
4 {{ /partial }}
```

Now, Antlers will ignore the `partial` tag on line 1 and create a tag pair from the opening tag on line 2 and the closing tag on line 4.

# 1.6 Variables and Custom Variables

Antlers variables can come from various sources, including tags, the Cascade, globals, and custom variables. Field handles and custom variable names can begin with a letter or underscore and may contain letters, numbers, underscores, and hyphens.

In the case of arrays, we can access specific array elements by using the path accessor syntax, which can be written using either a `.` or `:`. These two styles may be used interchangeably, except when working with tags.

**Figure 1.29**

```
{{
  _simple_array = [
    'One',
    'Two',
    'Three'
  ];

  _associative_array = [
    'city'  => 'Fargo',
    'state' => 'ND'
  ];
}}

{{# Outputs "Three" #}}
{{ _simple_array.2 }}
{{ _simple_array:2 }}


{{# Outputs "Fargo" #}}
{{ _associative_array.city }}
{{ _associative_array:city }}
```

## 1.7 Data Types

While Antlers mainly works on array output from tags and other sources, it does support many native data types such as booleans, numbers, strings, and arrays.

Boolean values are created by using either the `true` or `false` keywords:

**Figure 1.30**

```
{{ _truthy_value = true }}
{{ _falsey_value = false }}
```

Numeric values can be either integers or float values:

**Figure 1.31**

```
{{ _integer_value = 42 }}
{{ _float_value   = 10.10 }}
```

Arrays may be created by either using the `arr` operator or by using the shorthand bracket syntax:

**Figure 1.32**

```
{{ _simple_array     = arr(1, 2, 3) }}
{{ _simple_array_two = [1, 2, 3] }}
```

Strings may be created using single or double quotes and can span multiple lines:

**Figure 1.33**

```
{{

  _single_quote = 'Without hesitation, she nodded, knowing that
wherever he went, she would follow. They set off into the night,
hand in hand, not knowing what the future held, but filled with
hope and excitement for what was to come.';

  _double_quote = "Days turned into weeks, and weeks turned into
months, as they traveled across the country, exploring new places
and experiencing new adventures together. Along the way, they
faced many challenges and obstacles, but they always faced them
together, their love giving them the strength to overcome
anything that came their way.";

/}}
```

# 1.8 Assignment Operators

Assignment operators can be used to create new variables or update the value of existing variables.

## Left Assignment

The left assignment operator will set the variable reference on the left to the value appearing on the right. If the variable reference does not exist, a new variable will be created in the current scope.

**Figure 1.34**

```
{{# Creates a new variable. #}}
{{ _new_variable = 100 }}

{{# Update an existing variable. #}}
{{ _new_variable = 150 }}
```

## Addition Assignment

The addition assignment operator will sum the left-hand variable's value with the value on the right and update the variable reference to contain the new sum if the value on the right is numeric.

**Figure 1.35**

```
{{# Creates a new variable. #}}
{{ _total = 10 }}

{{# Increments the total by 5. #}}
{{ _total += 5 }}

{{# Outputs 15 #}}
{{ _total }}
```

When the right-hand value is a string type, the addition assignment operator will behave like a string concatenation operator. In the following example, we are using an empty string to coerce the right-hand value to a string:

**Figure 1.36**

```
{{ _value = ''; }}

{{ loop from="0" to="10" }}
  {{ _value += '' + count }}
{{ /loop }}

{{# Outputs 1234567891011 #}}
{{ _value }}

{{ _value = '' }}

{{ loop from="0" to="10" }}
  {{ _value += count }}
{{ /loop }}

{{# Outputs 66 #}}
{{ _value }}
```

The addition assignment operator will append the value on the right if the lefthand variable reference is an array.

## Subtraction Assignment

The subtraction assignment operator will subtract the value on the right from the value on the left and assign the difference to the variable reference on the left.

**Figure 1.37**

```
{{# Creates a new variable. #}}
{{ _total = 10 }}

{{# Decrements the total by 5. #}}
{{ _total -= 5 }}

{{# Outputs 5 #}}
{{ _total }}
```

## Multiplication Assignment

The multiplication assignment operator will multiply the variable's value on the left with the righthand value and assign the product to the lefthand variable reference.

**Figure 1.38**

```
{{# Creates a new variable. #}}
{{ _total = 10 }}

{{# Multiplies the total by 5. #}}
{{ _total *= 5 }}

{{# Outputs 50 #}}
{{ _total }}
```

## Division Assignment

The division assignment operator will divide the variable's value by the righthand value on the left and assign the quotient to the lefthand variable reference.

**Figure 1.39**

```
{{# Creates a new variable. #}}
{{ _total = 10 }}

{{# Divide the total by 5. #}}
{{ _total /= 5 }}

{{# Outputs 2 #}}
{{ _total }}
```

## Modulus Assignment

The modulus assignment operator will assign the remainder of the lefthand value divided by the righthand value to the lefthand variable reference.

**Figure 1.40**

```
{{ _total = 15 }}
{{ _total %= 2 }}

{{# Outputs 1 #}}
{{ _total }}
```

## Truthy Assignment (Gatekeeper)

The truthy assignment operator will evaluate the righthand expression if the lefthand value is truthy. The lefthand value is *not* updated to reflect the value of the righthand expression.

**Figure 1.41**

```
{{# Outputs Hello #}}
{{ true ?= 'Hello' }}

{{# No output #}}
{{ false ?= 'World' }}
```

# 1.9 String Concatenation

The string concatenation operator is used to combine two or more strings.

**Figure 1.42**

```
{{ _string_one = 'Hello' }}
{{ _string_two = 'world' }}

{{# Outputs Hello world #}}
{{  _string_one + ' ' + _string_two }}
```

# 1.10 String Concatenation Assignment

The concatenation assignment operator will append the righthand string value to the lefthand string value. There are two styles of string concatenation assignment, the first being the `.=` operator. The `.=` string concatenation assignment operator will coerce all runtime values to strings:

**Figure 1.43**

```
{{ _string_one = 'Hello' }}
{{ _string_two = 'world' }}

{{ _new_string = null }}

{{ _new_string .= _string_one }}
{{ _new_string .= ' ' + _string_two }}

{{# Outputs Hello world #}}
{{ _new_string}}
```

This is in contrast to the += style, which expects the lefthand value to already be a string in order to successfully concatenate the strings:

**Figure 1.44**

```
{{ _string_one = 'Hello' }}
{{ _string_two = 'world' }}

{{ _new_string = null }}

{{ _new_string += _string_one }}
{{ _new_string += ' ' + _string_two }}

{{# Outputs 0 #}}
{{ _new_string}}
```

In **Figure 1.44**, the starting value on line 4 was `null`, which will cause the concatenation to fail. However, if our starting value was a valid string, our concatenation will succeed:

**Figure 1.45**

```
{{ _string_one = 'Hello' }}
{{ _string_two = 'world' }}

{{ _new_string = '' }}

{{ _new_string += _string_one }}
{{ _new_string += ' ' + _string_two }}

{{# Outputs Hello world #}}
{{ _new_string}}
```

# 1.11 Logical and Comparison Operators

In this section, we explore the essentials of decision-making in Antlers and the various operators for comparing and evaluating expressions to control the flow of templates.

## Equality

The quality operator will return true if the left and righthand side values are equal. This operator will perform type coercion.

**Figure 1.46**

```
{{# Outputs Yes #}}
{{ if 1 == '1' }}
  Yes
{{ else }}
  No
{{ /if }}

{{ if storytime }}
Little did she know that he felt the same way about
```

```
her, and that together they would embark on a journey
of passion, adventure, and everlasting love. With each
passing moment, their bond grew stronger, until they
knew without a doubt that they were meant to be
together forever.
{{ /if }}
```

## Identity (Strict Equality)

The identity, or strict equality operator, will return true if the left and right-hand values are equal and have the same type. This operator does not perform type coercion of runtime types.

**Figure 1.47**

```
{{# Outputs No #}}
{{ if 1 === '1' }}
    Yes
{{ else }}
    No
{{ /if }}
```

## Greater Than

The greater than operator will return true if the righthand value is greater than the lefthand value.

**Figure 1.48**

```
{{# Outputs Yes #}}
{{ if 5 > 2 }}
  Yes
{{ else }}
  No
{{ /if }}
```

## Greater Than or Equal To

The greater than or equal to operator will return true if the righthand value is greater than *or* equal to the lefthand value.

**Figure 1.49**

```
{{# Outputs Yes #}}
{{ if 2 >= 2 }}
  Yes
{{ else }}
  No
{{ /if }}
```

## Less Than

The less than operator will return true if the righthand value is less than the lefthand value.

**Figure 1.50**

```
{{# Outputs No #}}
{{ if 5 < 2 }}
  Yes
{{ else }}
  No
{{ /if }}
```

## Less Than or Equal To

The less than or equal to operator will return true if the righthand value is less than *or* equal to the lefthand value.

**Figure 1.51**

```
{{# Outputs Yes #}}
{{ if 2 <= 2 }}
  Yes
{{ else }}
  No
{{ /if }}
```

## Not Equal

The not equal operator will return true if the righthand value is not equal to the lefthand value. This method will perform type coercion.

**Figure 1.52**

```
{{# Outputs No #}}
{{ if 5 != '5' }}
  Yes
{{ else }}
  No
{{ /if }}
```

## Inequality (Strict not-equal)

The inequality or strict not-equal operator will return true if the righthand value is not equal to the lefthand value. The lefthand and righthand values must be the same type; the inequality operator does not perform type coercion on runtime types.

**Figure 1.53**

```
{{# Outputs Yes #}}
{{ if 5 !== '5' }}
  Yes
{{ else }}
  No
{{ /if }}
```

## Spaceship

The Antlers spaceship operator behaves similarly to PHP's spaceship operator. This operator returns the following values:

- Returns `-1` if the lefthand value is less than the righthand value
- Returns `0` if the lefthand value is equal to the righthand value
- Returns `1` if the lefthand value is greater than the righthand value

## Logical AND

The logical AND operator is used to assert that two or more expressions are truthful. If any expression evaluates to false, the entire expression will return false. There are three styles of logical AND operator:

- The symbolic `&&` operator
- The symbolic `&` operator
- The `AND` keyword

### The `&` Operator

The symbolic `&` operator will perform the same operation as both `&&` and `AND` for backwards compatibility reasons.

**Figure 1.54**

```
{{ if true && true }}
  Yes
{{ /if }}

{{ if true & true }}
  Yes
{{ /if }}

{{ if true and true }}
  Yes
{{ /if }}
```

## Logical OR

The logical OR operator asserts that at least one expression is truthful. If an expression is false, the next expression will be evaluated. If all expressions return false, the entire expression will be false. There are two styles of OR operator:

- The symbolic `||` operator
- The `OR` keyword

**Figure 1.55**

```
{{ if false OR true }}
  Yes
{{ /if }}

{{ if false || true }}
  Yes
{{ /if }}
```

## Logical NOT

The NOT or negation operator will return the inverse of a truthy value that appears to the right of the operator. There are two styles of logical NOT operator:

- The symbolic `!` operator
- The `NOT` keyword

**Figure 1.56**

```
{{ if !false == true }}
  Yes
{{ /if }}

{{ if NOT false == true }}
  Yes
{{ /if }}
```

## Null Coalescence

The null coalescence operator will return the righthand value if the righthand value is null or undefined. Chained null coalescence operators will continue this behavior until the first non-null value is found; if no non-null value is encountered, no value will be returned.

**Figure 1.57**

```
{{# Outputs Default Title #}}
{{ _title ?? 'Default Title' }}

{{ _title = 'A Custom Title' }}

{{# Outputs A Custom Title #}}
{{ _title ?? 'Default Title' }}
```

# 1.12 Arithmetic Operators

In this section we dive into numerical calculations: understand the usage of addition, subtraction, multiplication, and division operators to perform complex calculations within your Antlers templates.

### Number Modifiers

Statamic provides a wide array of numeric modifiers, some of which are outlined in Appendix D: Number Modifier Reference.

## Addition

The addition operator will add the righthand value to the lefthand value.

**Figure 1.58**

```
{{# Outputs 10 #}}
{{ 5 + 5 }}
```

## Subtraction

The subtraction operator will reduce the lefthand value by the righthand value.

### Whitespace Required

Because hyphens are legal characters in variable names, there must always be whitespace on either side of the - subtraction operator.

**Figure 1.59**

```
{{# Outputs 0 #}}
{{ 5 - 5 }}
```

## Multiplication

The multiplication operator will multiply the lefthand value by the righthand value.

**Figure 1.60**

```
{{# Outputs 25 #}}
{{ 5 * 5 }}
```

## Division

The division operator will divide the lefthand value by the righthand value—division by zero results in an exception.

**Figure 1.61**

```
{{# Outputs 1 #}}
{{ 5 / 5 }}
```

## Modulo

The modulo operator produces the remainder of the lefthand value divided by the righthand value.

**Figure 1.62**

```
{{# Outputs 0 #}}
{{ 50 % 2 }}
```

## Exponentiation

The exponentiation operator raises the lefthand value to the righthand value.

**Figure 1.63**

```
{{# Outputs 10000000000 #}}
{{ 10 ** 10 }}
```

## Factorial

The factorial operator calculates the product of all positive integers less than or equal to the lefthand value.

**Figure 1.64**

```
{{# Outputs 24 #}}
{{ 4! }}
```

# 1.13 Bitwise Operators

Delve into binary-level manipulations: grasp the concept and application of bitwise operators to fine-tune your template logic and improve performance.

## Bitwise AND

The `bwa` keyword represents the bitwise AND operator. It is a binary operator, which means it operates on two operands - in this case, two numbers - and returns a new number as its result.

When you perform a bitwise AND operation on two numbers, the operation compares the binary representation of each number bit by bit. For each bit, if both numbers have a 1 in that position, the result will have a 1 in that position. Otherwise, the result will have a 0 in that position.

**Figure 1.65**

```
{{# Outputs 2 #}}
{{ 6 bwa 3 }}
```

In this example, we use the bitwise AND operator to compare the binary representation of the numbers 6 and 3. The binary representation of 6 is `110`, and the binary representation of 3 is `011`. When we perform the bitwise AND operation, we get the result `010`, the binary representation of the number 2.

## Bitwise OR

The bitwise OR operation is represented by the `bwo` keyword. When you perform a bitwise OR operation on two numbers, the operation compares the binary representation of each number bit by bit. For each bit, if at least one of the numbers has a 1 in that position, the result will have a 1 in that position. Otherwise, the result will have a 0 in that position.

**Figure 1.66**

```
{{# Outputs 7 #}}
{{ 6 bwo 3 }}
```

In this example, we use the bitwise OR operator to compare the binary representation of the numbers 6 and 3. The binary representation of 6 is `110`, and the binary representation of 3 is `011`. When we perform the bitwise OR operation, we get the result `111`, which is the binary representation of the number 7.

## Bitwise XOR

The bitwise XOR operator is represented by the `bxor` keyword. When you perform a bitwise XOR operation on two numbers, the operation compares the binary representation of each number bit by bit. For each bit, the result will have a 1 in that position only if one of the numbers has a 1 in that position, but not both. Otherwise, the result will have a 0 in that position.

**Figure 1.67**

```
{{# Outputs 5 #}}
{{ 6 bxor 3 }}
```

In this example, we use the bitwise XOR operator to compare the binary representation of the numbers 6 and 3. The binary representation of 6 is `110`, and the binary representation of 3 is `011`. When we perform the bitwise XOR operation, we get the result `101`, which is the binary representation of the number 5.

## Bitwise NOT

The bitwise NOT operator is represented by the `bnot` keyword; it operates on the operand that appears to its right. When you perform a bitwise NOT operation on a number, the operation inverts each bit of the binary representation of that number. In other words, every 0 becomes a 1, and every 1 becomes a 0.

**Figure 1.68**

```
{{# Outputs -7 #}}
{{ bnot 6 }}
```

In this example, we use the bitwise NOT operator to invert the binary representation of the number 6. The binary representation of 6 is `110`, and when we perform the bitwise NOT operation, we get the result `11111111111111111111111111111001`. This result is negative because PHP uses two's complement notation to represent signed integers.

## Bitwise Shift Left

The bitwise shift left operator is represented by the `bsl` keyword. When you perform a bitwise shift left operation on a number, the operation shifts the binary representation of that number to the left by a specified number of positions. In other words, it adds zeros to the right-hand side of the binary representation of the number.

**Figure 1.69**

```
{{# Outputs 24 #}}
{{ 6 bsl 2 }}
```

In this example, we use the bitwise shift left operator to shift the binary representation of the number `6` two positions to the left. The binary representation of `6` is `110`, and when we shift it to two positions to the left, we get the result `11000`, which is the binary representation of the number `24`.

## Bitwise Shift Right

The bitwise shift right operator is represented by the `bsr` keyword. When you perform a bitwise shift right operation on a number, the operation shifts the binary representation of that number to the right by a specified number of positions. In other words, it removes digits from the right-hand side of the binary representation of the number.

**Figure 1.70**

```
{{# Outputs 3 #}}
{{ 6 bsr 1 }}
```

In this example, we use the bitwise shift right operator to shift the binary representation of the number `6` one position to the right. The binary representation of `6` is `110`, and when we shift it one position to the right, we get the result `11`, which is the binary representation of the number `3`.

# 1.14 Antlers Style Guide

This section contains a few recommendations to help encourage consistency when working on Statamic sites and to avoid common issues. The topics covered in this section are merely suggestions and can be adapted to your coding style.

## Hyphens in Variable Names

When creating handles or variables, use underscores over hyphens.

Instead of:

**Figure 1.71**

```
{{ _variable-name = 'some value' }}
```

Prefer:

**Figure 1.72**

```
{{ _variable_name = 'some value' }}
```

## Custom Variable Names

Custom variable names should be prefixed with a single underscore; this helps to recognize what variables are custom and which came from other sources, such as tags or the Cascade:

**Figure 1.73**

```
{{ _custom_title = 'My Title' }}
```

Additionally, using `_snake_case` for custom variable names is preferred.

## Define Arrays Outside of Loops

Unless an array contains dynamic data that can only be set from within a loop, all custom arrays should be created outside of loops for performance reasons.

Instead of:

**Figure 1.74**

```
{{ loop from="0" to="10" }}
  {{ _custom_array = [1,2,3] /}}

  {{ if _custom_array | contains(value) }}
    Yes.
  {{ /if }}
{{ /loop }}
```

Prefer:

**Figure 1.75**

```
{{ _custom_array = [1,2,3] /}}

{{ loop from="0" to="10" }}

  {{ if _custom_array | contains(value) }}
    Yes.
  {{ /if }}
{{ /loop }}
```

## Self-Closing Tags

Make all tags self-closing when they should not become part of a tag pair. This should be done when there is a tag pair within the same template with the same name as a non-tag pair.

Instead of:

**Figure 1.76**

```
{{ title | upper }}
{{ _my_variable = 'A value' }}

{{ _my_variable | explode(' ') }}
  {{ value }}
{{ /_my_variable }}
```

Prefer:

**Figure 1.77**

```
{{ title | upper }}
{{ _my_variable = 'A value' /}}

{{ _my_variable | explode(' ') }}
  {{ value }}
{{ /_my_variable }}
```

## Use of Semi-Colons

Do not end expressions with a semicolon unless multiple expressions are within a single Antlers tag.

Instead of:

**Figure 1.78**

```
{{ _my_variable     = 'A value'; }}
{{ _second_variable = 'one' _third_variable = 'two' }}
```

Prefer:

**Figure 1.79**

```
{{ _my_variable     = 'A value' }}
{{ _second_variable = 'one'; _third_variable = 'two'; }}
```

## Modifier Syntax

Always use the pipe modifier syntax, even for tag pairs.

Instead of:

**Figure 1.80**

```
{{ array_field reverse="true" }}

{{ /array_field }}
```

Prefer:

**Figure 1.81**

```
{{ array_field | reverse }}

{{ /array_field }}
```

# 2. Partial Templates

Statamic supports the concepts of partials or sections of more extensive templates that we can use inside more significant templates, such as the site's overall layout or within templates for specific pages. Partial templates are beneficial to organizing a website template in several ways:

1. **Modular structure**: Partial templates allow you to break down a website template into smaller, modular pieces. Breaking our templates into smaller pieces makes managing and updating your website design more manageable, as you can work on individual components without modifying the entire template.
2. **Reusability**: We can use partial templates across different pages of your website, saving time and effort in development. A commonly used example is the concept of headers and footers, which we can solve quickly using a layout within Statamic. However, we can use partial templates to effortlessly swap out these components for specific pages, such as the home page or an event's landing page.
3. **Code organization**: Partial templates help organize your code logically, making it easier to navigate and maintain. You can group related code into separate partials, making finding and modifying specific template sections easier. Applying this concept to certain fieldtypes, such as the replicator or Bard fieldtype, allows us to manage complex designs while providing a friendly control panel experience for our clients or site editors.
4. **Collaboration**: When working in a team, we can leverage partials to make it easier for multiple developers to implement design components simultaneously. Additionally, utilizing a component mindset when constructing our partials makes it easier to leverage and share individual partial templates with the wider Statamic community, with Peak, a starter kit by Studio 1902, an exemplary example.

Overall, partial templates can make website development more efficient and organized. By breaking down a website template into smaller, reusable pieces, you can simplify the development process and improve the overall quality of your code.

As a simple example, let us consider the following template:

**Figure 9.1**

```
Before

{{ partial:test }}

After
```

Our template in **Figure 9.1** uses the `partial` Tag to include a file named `test.antlers.html`. If you are familiar with Laravel's Blade templating engine, the `partial` Tag behaves similarly to the `include` directive. Now, if we were to create a file named `_test.antlers.html` with the following contents:

**Figure 9.2**

```
The partial contents.
```

We would be able to see the following results in our browser:

**Figure 9.3**

```
Before

The partial contents.

After
```

This example is not particularly interesting but demonstrates the basics of using partial templates. What's interesting about our example is how we named our partial file; our file name begins with the _ character. This is *not* a requirement, but it will prevent that file from appearing as an option in any template fieldtype within the control panel.

## 2.1 Partials and Frontmatter

Frontmatter is a block of YAML that can appear at the top of our template files, including partials. These values are added to an array named `view` that we can use to access specific values. For instance, the following partial:

**Figure 9.4**

```
---
color: blue
---

The color is {{ view:color }}
```

Would produce the following results:

**Figure 9.5**

```
The color is blue
```

A lesser-known fact is that partial parameters are also added to the `view` array, and their value will override any value that already exists. We can use this to our advantage to allow customization of values but also manage defaults in a single place.

Let's create a simple alert partial that allows for the customization of the type while also providing a default.

In `_alert.antlers.html`:

**Figure 9.6**

```
---
type: primary
---

<div class="alert alert-{{ view:type }}" role="alert">
  {{ slot }}
</div>
```

Including our partial like so:

**Figure 9.7**

```
{{ partial:alert }}
  The message.
{{ /partial:alert }}

{{ partial:alert type="success" }}
  The success message.
{{ /partial:alert }}
```

Produces the following end result:

**Figure 9.8**

```
<div class="alert alert-primary" role="alert">
  The message.
</div>

<div class="alert alert-success" role="alert">
  The success message.
</div>
```

When we do not supply a `type` parameter to our alert partial, the default value of `primary` is returned. However, when we provide a parameter of the same name, its value is used instead. This example works well for constructing components for

frameworks like Bootstrap, and it may need to be clarified how we might utilize the same technique for other CSS frameworks, such as Tailwind CSS.

When attempting to implement something similar using a framework that primarily works using utility classes, the primary challenge is how to associate a friendly parameter value with the class list cleanly. For example, a common technique I've seen used is to use an inline condition to determine which value we should use:

**Figure 9.9**

```
---
styles:
  primary: 'text-blue-500'
  success: 'text-green-500'
  danger: 'text-red-500'
---

<div class="{{ view:styles[{type ?? 'primary'}] }}" role="alert">
  {{ slot }}
</div>
```

While this certainly works for simple partial templates, it can quickly become tedious and error-prone as the number of places our variable needs to be utilized increases. We can overcome this by creating a temporary variable at the top of our partial like so:

**Figure 9.10**

```
---
styles:
  primary: 'text-blue-500'
  success: 'text-green-500'
  danger: 'text-red-500'
---

{{ _type = type ?? 'primary' }}

<div class="{{ view:styles[_type] }}" role="alert">
```

```
  {{ slot }}
</div>
```

But over time, this can also become cluttered, and it may not be easy to quickly understand what is going on for any new developers coming onto a project.

Instead, I like to use a second frontmatter value to contain the type value, which we can then use to specify a default and look up our additional styles later. As an example, let us consider the following alert partial that uses the `type` parameter to customize the class list that appears within the alert's container element, as well as its text wrapper:

**Figure 9.11**

```
---
type: primary
container_styles:
  primary: 'p...'
  success: 's...'
  danger: 'd...'
text_styles:
  primary: 'text-blue-500'
  success: 'text-green-500'
  danger: 'text-red-500'
---

<div class="{{ view:container_styles[view:type] }}" role="alert">
  <span class="{{ view:text_styles[view:type] }}">
    {{ slot }}
  </span>
</div>
```

Our frontmatter approach now allows us to manage our class list in a single location while providing default values close to where those class lists are defined.

A fair criticism may be that all of our variables are now prefixed with `view`, and do not read like the parameters we supplied. This concern can be resolved by wrapping

the contents of our partial template in a `view` tag pair, which will provide us access to all of our usual parameters with default options applied:

**Figure 9.12**

```
---
type: primary
container_styles:
  primary: 'p...'
  success: 's...'
  danger: 'd...'
text_styles:
  primary: 'text-blue-500'
  success: 'text-green-500'
  danger: 'text-red-500'
---

{{ view }}
  <div class="{{ container_styles[type] }}" role="alert">
          <span class="{{ text_styles[type] }}">
      {{ slot }}
    </span>
  </div>
{{ /view }}
```

## 2.2 Working with Slots

We've already seen the concepts of slots in use through the alert example. When we use the `partial` Tag as a tag pair, the content that appears between the opening and closing tags will be made available through the `slot` variable:

**Figure 9.13**

```
{{ partial:filename }}
  This content will be available in the slot variable.
{{ /partial:filename }}
```

We can display that content inside our partial using `{{ slot }}`. An important thing to remember is *slots are variables*. Because slots are variables, other language features, such as modifiers, are also available.

As an example, if we wanted to ensure the slot's content does not contain line breaks or has its extraneous whitespace removed, we can use the `collapse_whitespace` modifier:

## Leading/Trailing Whitespace

If you are only concerned about removing the leading/trailing whitespace from a slot's content, this is automatically removed for you.

**Figure 9.14**

```
{{#
  Render the slot contents
  with collapsed whitespace.
#}}
<span>{{ slot | collapse_whitespace }}</span>
```

Because slot content is made available as a variable, we can also check if it has been supplied using conditions. An important thing to note is that if slot contents are composed entirely of whitespace, they are *not* added to the `slot` variable:

**Figure 9.15**

```
{{ if slot }}
  Yes: {{ slot | upper }}
{{ else }}
  Default content.
{{ /if }}
```

Our example in **Figure 9.15** would produce the following output; note that only the second `partial` Tag pair caused our condition to be `true`:

**Figure 9.16**

```
Default content.

Yes: SOME CONTENT.
```

Antlers also supports *named slots*, allowing us to receive and display content at arbitrary locations within our partial template. Like normal slot content, named slots are also made available as variables within our partial so we can check for their existence and use modifiers like usual:

In `_card.antlers.html`:

**Figure 9.17**

```
<div class="...">
  {{ if slot:title }}
    {{ slot:title }}
  {{ else }}
    <h3>{{ title }}</h3>
  {{ /if }}

  <p>{{ slot | collapse_whitespace }}</p>
</div>
```

## Named Slot Variables

The `slot` variable system is unique within Antlers. While it shares the same syntax as arrays (via. `slot:title`), the contents of the `slot` variable cannot be accessed as an array inside a template.

Including our partial from **Figure 9.17** like so:

**Figure 9.18**

```
{{ partial:card }}
  {{ slot:title }}
    <h4>A custom title.</h4>
  {{ /slot:title }}

  The content.
{{ /partial:card }}

{{ partial:card }}
  The content.
{{ /partial:card }}
```

Produces the following output, assuming the page's title was "Home":

**Figure 9.19**

```
<div class="...">
  <h4>A custom title.</h4>

  <p>The content.</p>
</div>

<div class="...">
  <h3>Home</h3>

  <p>The content.</p>
</div>
```

## 2.3 Managing Partial Tag Pairs

When working with multiple partials, it is common to have many Tags close together like so:

**Figure 9.20**

```
{{ partial src="one" }}
{{ partial src="two" }}
  <!-- Slot content -->
{{ /partial }}
```

The tag pairing algorithm in this situation will pair the first `partial` Tag to the final closing tag, causing our second `partial` Tag to be nested inside. We can force Antlers to create a tag pair out of the second `partial` Tag by making the first one self-closing:

**Figure 9.21**

```
{{ partial src="one" /}}
{{ partial src="two" }}
  <!-- Slot content -->
{{ /partial }}
```

By making the first tag self-closing, Antlers will pair the second and third Antlers tags instead of having the second `partial` Tag become part of the slot content for the "one" partial.

## 2.4 Dynamically Loading Partials

We can dynamically load partial templates using variable interpolation combined with the `src` parameter, a common technique for sites using the page builder pattern. As an example, let us take a look at the following template, which loads different partials templates depending on the set within a replicator field:

**Figure 9.22**

```
{{ field_name }}
  {{ if type == 'text' }}
    {{ partial:components/text /}}
  {{ elseif type == 'image' }}
    {{ partial:components/image /}}
  {{ elseif type == 'table' }}
    {{ partial:components/table /}}
  {{ /if }}
{{ /field_name }}
```

The conditions can be refactored away using string interpolation within the `src` parameter:

**Figure 9.23**

```
{{ field_name }}
  {{ partial src="components/{type}" /}}
{{ /field_name }}
```

The result will be the same, with much less template code to maintain. As the site project continues, if a new set is added to some content that does not have a corresponding partial, we will receive an error similar to the following:

**Figure 9.24**

```
View [button] not found.
```

Depending on the context, this may be beneficial. However, we can use the `partial:if_exists` Tag to only attempt to load the partial template if it is available:

**Figure 9.25**

```
{{ field_name }}
  {{ partial:if_exists src="components/{type}" /}}
{{ /field_name }}
```

# 2.5 Recursive Partials

In web development, recursive partials refer to a technique of using partial templates that contain themselves as a part of their rendering logic. This means that a partial template can call itself to render a nested section of its data structure. Recursive partials can be useful in situations where the data to be displayed has a hierarchical or nested structure, and the design requirements call for displaying that data in a visually nested way.

One common example of where recursive partials may be needed is when displaying comment feeds. Comments often have a hierarchical structure where a reply to a comment is displayed nested under its parent comment. In this case, the comment feed partial template could be designed to call itself recursively to render each nested level of comments.

## Maintainability of Recursive Partials

It is important to note that while recursive partials can be useful in certain situations, they can also be complex and difficult to manage. Careful consideration should be given to the design and implementation of recursive partials to ensure they are efficient, maintainable, and don't lead to performance issues or infinite loops.

Let us consider the following sample comment data:

**Figure 9.26**

```
comments:
  -
    content: 'The first comment'
    replies:
      -
        content: 'The first reply'
        replies: {  }
      -
        content: 'The second reply'
        replies: {  }
      -
        content: 'The third reply'
        replies:
          -
            content: 'A nested reply'
            replies: {  }
  -
    content: 'The second comment'
    replies: {  }
  -
    content: 'The third comment'
    replies: {  }
```

We will create a `comment` partial to render each of our comments. To start the process of rendering our comment feed, we need to start rendering the initial list:

**Figure 9.27**

```
<ul class="root list">
  {{ comments }}
    {{ partial:comment /}}
  {{ /comments }}
</ul>
```

If we take a moment to look at our sample data, we can see that each of the comments has a `replies` array, even though some are empty. We must have some way of

existing our recursive partial. Otherwise, we will end up in an infinite loop. In our case, we will check if the `replies` contains any items:

**Figure 9.28**

```
<li>
  <p>Content: {{ content }}</p>

  {{ if replies }}
  <ul class="nested list">
    {{ replies }}
      {{ partial:comment /}}
    {{ /replies }}
  </ul>
  {{ /if }}
</li>
```

Our recursion happens between lines 4 and 10; we can wrap the results in a custom wrapper element each time we begin rendering a recursive list. Once our site has been rendered, we will receive the following output:

**Figure 9.29**

```
<ul class="root list">
  <li>
    <p>Content: The first comment</p>

    <ul class="nested list">
      <li>
        <p>Content: The first reply</p>
      </li>

      <li>
        <p>Content: The second reply</p>
      </li>

      <li>
        <p>Content: The third reply</p>
```

```
16
17          <ul class="nested list">
18            <li>
19              <p>Content: A nested reply</p>
20            </li>
21          </ul>
22        </li>
23      </ul>
24    </li>
25
26    <li>
27      <p>Content: The second comment</p>
28    </li>
29
30    <li>
31      <p>Content: The third comment</p>
32    </li>
33  </ul>
```

Our template will now recursively render our comments, but, at the moment, it isn't easy to know our current recursive depth. We can solve this by making use of a custom `_depth` variable. To begin our implementation, we will modify our main template to initialize our `_depth` variable each time we render a new comment:

**Figure 9.30**

```
1  <ul class="root list">
2    {{ comments }}
3      {{ _depth = 0; }}
4        {{ partial:comment /}}
5      {{ _depth = null; }}
6    {{ /comments }}
7  </ul>
```

Resetting our `_depth` variable to `null` on line 6 is not always necessary. Still, it does help to ensure our custom variable doesn't leak anywhere it shouldn't if we were to

modify or refactor our main loop in the future. With the initialization out of the way, we can now move on to incrementing our depth each time we recursively include our partial:

**Figure 9.31**

```
<li>
  <p>Current depth: {{ _depth }}</p>
  <p>Content: {{ content }}</p>
  {{ if replies }}
    {{ _depth += 1; }}

    <ul class="nested list">
      {{ replies }}
          {{ partial:comment /}}
      {{ /replies }}
    </ul>
    {{ _depth -= 1; }}
  {{ /if }}
</li>
```

Each time we are about to include our partial again, we increment the current depth by one on line 5. To ensure that any other items after our recursive list are rendered with the appropriate depth, we need also decrement the current value on line 12 each time we leave our recursive partial.

With our updated changes, our output would now be:

**Figure 9.32**

```
<ul class="root list">
  <li>
    <p>Current depth: 0</p>
    <p>Content: The first comment</p>

    <ul class="nested list">
      <li>
        <p>Current depth: 1</p>
        <p>Content: The first reply</p>
      </li>

      <li>
        <p>Current depth: 1</p>
        <p>Content: The second reply</p>
      </li>

      <li>
        <p>Current depth: 1</p>
        <p>Content: The third reply</p>

        <ul class="nested list">
          <li>
            <p>Current depth: 2</p>
            <p>Content: A nested reply</p>
          </li>
        </ul>
      </li>
    </ul>
  </li>

  <li>
    <p>Current depth: 0</p>
    <p>Content: The second comment</p>
  </li>

```

```
36    <li>
37      <p>Current depth: 0</p>
38      <p>Content: The third comment</p>
39    </li>
40  </ul>
```

## 2.6 Managing Custom Fieldsets with Partials

A common design pattern when building s to create reusable fields across different collections, forms, or pages. Additionally, it is also not uncommon to see specific partials created to display these fieldsets. This is nice since it allows for the same template code and groups of fields to be managed in a central location, and helps to keep things well organized and clean.

As an example, we could have the following fieldset to manage Frequently Asked Questions:

**Figure 9.33**

```
title: FAQ
fields:
  -
    handle: faq
    field:
      collapse: false
      sets:
        questions:
          display: Questions
          fields:
            -
              handle: question
              field:
                input_type: text
                antlers: false
                display: Question
                type: text
```

```
                icon: text
                listable: hidden
                instructions_position: above
                read_only: false
            -
              handle: answer
              field:
                antlers: false
                display: Answer
                type: textarea
                icon: textarea
                listable: hidden
                instructions_position: above
                read_only: false
      display: Questions
      type: replicator
      icon: replicator
      listable: hidden
      instructions_position: above
      read_only: false
```

We could create a partial named `components/faq.antlers.html` with the following template that would render our fieldset:

**Figure 9.34**

```
<dl>
  {{ faq }}
  <dt>{{ question }}</dt>
  <dd>{{ answer }}</dd>
  {{ /faq }}
</dl>
```

and simply include the following in other templates when we wanted to reuse our Frequently Asked Questions template:

**Figure 9.35**

```
{{ partial:components/faq /}}
```

## Working with Fieldset Prefixes

The previous example works great when field names do not change. However, because fieldsets can be prefixed, it becomes difficult to create truly reusable partials when targeting fieldsets (or when you may want to render the same group of fields multiple times on the same page, each with a different prefix). Let's assume that the previous fieldset was imported twice within the same blueprint. Once without a prefix, and a second time with the `product_` prefix:

**Figure 9.36**

```
title: Home
sections:
  main:
    display: Main
    fields:
      -
        handle: title
        field:
          type: text
          required: true
          character_limit: 0
          display: Title
          validate:
            - required
      -
        import: faq
      -
        import: faq
        prefix: product_
```

Our page would now contain a `faq` and `product_faq` variable, each with their own list of Frequently Asked Questions. With this setup, we could either duplicate our partial or simply alias the single variable to render both:

**Figure 9.37**

```
{{ partial:components/faq /}}

{{ partial:components/faq :faq="product_faq" /}}
```

This technique works well when the imported fieldsets are small (such as in this example), but can quickly become tedious if the imported fieldset contains many different fields. This often leads to attempts to writing invalid Antlers code like the following:

In `resources/views/faq.antlers.html`:

**Figure 9.38**

```
<dl>
  {{ {prefix}faq }}
    <dt>{{ question }}</dt>
    <dd>{{ answer }}</dd>
  {{ /{prefix}faq }}
</dl>

{{# In another file. #}}
{{ partial:components/faq prefix="product_" /}}
```

Even if the above had worked, it would make handling the non-prefixed case much more difficult. To help with this situation, Antlers supports the concept of forcing the template engine to check if a variable exists with any given prefix, before falling back to the normal cascade rules.

## Fieldset Prefixes and Variable Names

To have the Antlers check if variables exist with a prefix, you simply need to add the `handle_prefix` parameter to either the `partial` Tag or the `scope` Tag:

**Figure 9.39**

```
{{ partial:components/faq /}}

{{ partial:components/faq handle_prefix="product_" /}}
```

When the Antlers encounters the `handle_prefix` parameter, every variable will be checked to see if it exists with a prefix (but only within the partial or scope Tag on which it was applied).

When the above code is being evaluated, it would be as if we had written this Antlers instead:

**Figure 9.40**

```
<dl>
  {{ faq }}
  <dt>{{ question }}</dt>
  <dd>{{ answer }}</dd>
  {{ /faq }}
</dl>

<dl>
  {{ product_faq }}
  <dt>{{ question }}</dt>
  <dd>{{ answer }}</dd>
  {{ /product_faq }}
</dl>
```

The first partial call would render like normal, because no prefix was specified. However, in the second partial call we are specifying the `product_` prefix. Because a `product_faq` variable *does* exist, that will be used instead of the `faq` variable (if no prefixed match was found, Antlers will revert back to normal variable Cascade logic).

The `scope` Tag version might look something like this:

**Figure 9.41**

```
{{ scope handle_prefix="product_" }}
  <dl>
    {{ faq }}
    <dt>{{ question }}</dt>
    <dd>{{ answer }}</dd>
    {{ /faq }}
  </dl>
{{ /scope }}
```